when a virtual environment has been created and there is the need to create a requirements.txt file, in which directory should the…

To do, what’s left on CAL\_BridgeData:

1. Can resampling be used to reduce the number of observations in a dataset?
2. Set up and use virtual environment on the project
3. Split up the functions/classes in the single program file (burns my ass but I may have to leave things in a single file- and just beg forgiveness from MAXX…)
4. Make the Map of the Bridge locations
   1. Think of how to make the latitudes and longitudes matter in terms of effect on CS1-CS4.
   2. Include in the analysis being presented to MAXX? Probably not/
5. Look at articles to use loc/iloc correctly
   1. Same for variable scope,
   2. Same for immutable objects in arguments
   3. ….????

def parse\_XML(xml\_file, df\_cols):

xtree = et.parse(xml\_file)

xroot = xtree.getroot()

rows = []

for node in xroot:

res = []

res.append(node.attrib.get(df\_cols[0]))

for el in df\_cols[1:]:

if node is not None and node.find(el) is not None:

res.append(node.find(el).text)

else:

res.append(None)

rows.append({df\_cols[i]: res[i]

for i, \_ in enumerate(df\_cols)})

out\_df = pd.DataFrame(rows, columns=df\_cols)

return out\_df

# Parse the XML files for the individual years (Important to keep the tags in the original files the same in the parse commands or the data at those locations may come in as None or NaN!)

import glob

files = glob.glob('\*.xml')

print(files)

# Split filename at the underscore (\_) creates a year with the state abbreviation after

# Then follow on by adding df to the front of the string to act as variables names for the different DataFrames.

df\_names = ["df" + i.split('\_', 1)[0] for i in files]

root= "./CAL\_BridgeData"

#container for the various xmls contained in the directory

# collect xml filenames and paths

for dirpath, dirnames, filenames in os.walk(root):

for file in files:

files.append(dirpath + '\\' + file)

# end collection of xml filenames and paths

df\_cols = "FHWAED", "STATE", "STRUCNUM", "EN", "EPN", "TOTALQTY", "CS1", "CS2", "CS3", "CS4"

# Begin parsing of dataframes recursively

# create empty list for dataframes

dataframes = []

for i in range(len(df\_names)):

temp\_df = parse\_XML(files[i], df\_cols)

dataframes.append(temp\_df) # creates a list of dataframes with the contents of the xml files read into them.

for z in range(len(files)):

dataframes[z]['filename'] = files[z]

# !!!

# df

for df in dataframes:

# Print the data types of the columns

print("DataFrame:", df)

print("Column Data Types:")

print(df.dtypes)

print()

# Begin data type conversion

# Define the desired data types for conversion

desired\_data\_types = {

'STATE': 'str',

'STRUCNUM': 'str',

'EN': 'str',

'TOTALQTY': 'float',

'CS1': 'float',

'CS2': 'float',

'CS3': 'float',

'CS4': 'float',

'filename': 'str'

}

# Iterate over the dataframes

# !!!

# df

for df in dataframes:

# Convert data types of columns

df = df.astype(desired\_data\_types)

# Print the updated DataFrame with converted data types

print(df)

print()

# End data type conversion

# !!!

# !!!

# df\_nameToDF is the dictionary of dataframes as created when the df\_names are matched up with the data corresponding to the xml file from which the data is read at parse.

df\_nameToDF = {df\_names[x]:dataframes[x]for x in range(len(df\_names))}

# creates the dictionary of keys in the form of the df\_names list and the values in the form of the dataframes list. In this case for the state of California the keys will take the form df2016CA, df2017CA, ... df2022CA.

# bridge\_counts\_un means number of unique bridges or STRUCNUM in each df, performing this action to get a sense of the number of unique bridges in each data set.

# !!!

# df

bridge\_counts\_un = {k: df.groupby('STRUCNUM') for k, df in df\_nameToDF.items()}

# End determination of number of unique bridges

#df2016CA.groupby('STRUCNUM').count()

"""

df2016CA.groupby('STRUCNUM').count()

# 7578

df2017CA.groupby('STRUCNUM').count()

# 10019

df2018CA.groupby('STRUCNUM').count()

# 10439

df2019CA.groupby('STRUCNUM').count()

# 10851

df2020CA.groupby('STRUCNUM').count()

# 10873

df2021CA.groupby('STRUCNUM').count()

# 10877

df2022CA.groupby('STRUCNUM').count()

# 10899

df2023CA.groupby('STRUCNUM').count()

# 10896

"""

# Begin df\_nameToDF procedure, i.e. making the variable name with a nomenclature of df20XXCA where the XX is like 16 for 2016, 17 for 2017, etc.

# creates the individual dataframes with names corresponding to df\_names and the associated data in the form of a dataframe corresponding to the list called dataframes.

for k, v in df\_nameToDF.items():

vars()[k] = v

# End df\_nameToDF procedure

# Looked for a means of merging the dataframes within the dict of dataframes- without having to separate them out into individual variables- it may be possible, MVP II.

# In other words a lot of what happens below all the way to the area where the slicing of the concatenated dataframe begins is all meant to insure that the STRUCNUM for each year match each other and that random bridges with no matching bridge number in the other years and their associated data aren't being included in the data I intend to analyze.

# Inside the bridge\_array\_dict b\_16 thru b\_23 just mean "bridge number" aka STRUCNUM for the corresponding years- 2016 thru 2023, making a key that holds the list of STRUCNUM as an array for each year as it would be right after being parsed into a dataframe. In other words the b\_16 - b\_23 variables will be larger in size (i.e. no. of rows) than the dataframes as will be seen below once the STRUCNUM not present in all years are removed.

# Copy and then modify the existing dictionary using the existing keys to create new keys and copy the STRUCNUM column of the dataframe to the new dict and convert it to a numpy array.

bridge\_arr\_dict = {'b\_' + key[4:]: df['STRUCNUM'].to\_numpy() for key, df in df\_nameToDF.items()}

# !!! THIS IS SUBJECT TO CHANGE: The assumption I will use is that the data for all bridges (an individual bridge is denoted by STRUCNUM) that are common to all the years of data being analyzed (2016 - 2023 in this case) is to be considered, meaning that the data (condition states of individual bridge elements) associated with the bridges common across 8 years shall be used even if the data provided for a bridge one year is not provided for all the other years or is provided sporadically for other years (i.e. if the bridge components (EN) rated for condition state one year are not rated for all years - BUT some components of said bridge are rated for all years being considered and analyzed then those condition states for those elements can be used as part of the data). For instance, a very common bridge component (or EN, element number) is a deck constructed of reinforced concrete, which I refer to as a variable as deck\_rc, and this EN is number 12 as denoted by the Federal Highway Administration (FHWA) Specification for the National Bridge Inventory, Bridge Elements. As such it may be that the number of observations across the years considered is not the same for that element number (EN) each year- some years may include condition states associated with that element present for that bridge in some but not all years, but it is my intention to use as many observations as possible for as many bridge parts as possible to attempt to make the computer model accurate. Again, this is subject to change.

"""

df2016CA

# 56275 lines long

df2017CA

# 75574 lines long

df2018CA

# 78832 lines long

df2019CA

# 82570 lines long

df2020CA

# 83627 lines long

df2021CA

# 83933 lines long

df2022CA

# 85926 lines long

df2023CA

# 85926 lines long

"""

# Looking at how many occurrences of a single bridge occur in each year so that the total possible number of bridges being observed is accurate.

# !!!

# Here

# !!!

# the dfs were creating more entries than were present originally due to additional entries created when EN = EPN that also had entries for CS1-CS4 data as well \*\*\* I still need to explain to myself why this is necessary in order to avoid multiple entries of the same element for a single bridge.

""" the ...EPN.isnull() expressions below are required to merge the data properly while avoiding multiple entries of the same EN for each bridge """

"""

column\_name = 'EN'

dataframes = [df.dropna(subset=[column\_name]) for df in dataframes]

"""

"""

null\_checks = [df[column\_name].isnull() for df in dataframes]

print(null\_checks[0]) # Null checks for Year 2016 STRUCNUM

print(null\_checks[1]) # Null checks for Year 2017 STRUCNUM

print(null\_checks[2]) # Null checks for Year 2018 STRUCNUM

print(null\_checks[3]) # Null checks for Year 2019 STRUCNUM

print(null\_checks[4]) # Null checks for Year 2020 STRUCNUM

print(null\_checks[5]) # Null checks for Year 2021 STRUCNUM

print(null\_checks[6]) # Null checks for Year 2022 STRUCNUM

print(null\_checks[7]) # Null checks for Year 2023 STRUCNUM

"""

# Come up with the max number of bridges that are common to all years (first merge??)

# Come up with the highest total number of possible EN for all STRUCNUM?

# MVP II is the Minimum Viable Product version II- or a program/data analysis that would supercede this one.

# If I refer to "MVP II" and then comment out some code in that area I am referring to a functionality I have not achieved in this program and I would hope to make possible in a second version of this application were it to be updated.

# MVP II: Apply isnull() method to the EPN column of the dataframes while the dataframes are still in a list or dictionary and before any other changes are made to the list, rather than applying isnull() as I do below to each dataframe hard-coded manually.

filtered\_df\_nameToDF = {df\_name: df[df['EPN'].isnull()] for df\_name, df in df\_nameToDF.items()}

# change column suffixes to avoid confusion when merging at a later time in the program.

# change dataframe columns by adding suffix to avoid MergeErrors in future versions and to enhance readability.

# df2016CA [index:length] (index starts at zero, length starts at 1) so index = 2 means '2' then length of 6 means 6 in this case, so 2016.

def add\_suffix\_to\_col\_hdr(dict\_of\_dfs, cols\_to\_exclude = None):

for key, df in dict\_of\_dfs.items():

# Extract a part of the key to use as a suffix

suffix = key[2:6] # Making the suffixes something like CS1\_2016CA from what would have been CS1

# Modify columns by adding the suffix

df.columns = [col + '\_' + suffix if col not in cols\_to\_exclude else col for col in df.columns]

# Call the function to add suffixes to columns

add\_suffix\_to\_col\_hdr(filtered\_df\_nameToDF, cols\_to\_exclude = ['STRUCNUM', 'EN'])

# Print modified dataframes

for key, df in filtered\_df\_nameToDF.items():

print(f"{key}:\n{df}")

# 12.24.2023 revisions good to above.

#!!!

"""

df2016CA=df2016CA[df2016CA.EPN.isnull()]

# Drops the number of lines from 56275 to 50426

df2017CA=df2017CA[df2017CA.EPN.isnull()]

# Drops the number of lines from 75574 to 67593

df2018CA=df2018CA[df2018CA.EPN.isnull()]

# Drops the number of lines from 78832 to 70351

df2019CA=df2019CA[df2019CA.EPN.isnull()]

# Drops the number of lines from 82570 to 73470

df2020CA=df2020CA[df2020CA.EPN.isnull()]

# Drops the number of lines from 83627 to 74275

df2021CA=df2021CA[df2021CA.EPN.isnull()]

# Drops the number of lines from 83933 to 74585

df2022CA=df2022CA[df2022CA.EPN.isnull()]

# Drops the number of lines from 85926 to 75687

df2023CA=df2023CA[df2023CA.EPN.isnull()]

# Drops the number of lines from 86474 to 76053

"""

# Begin determine the set of STRUCNUM common to all years observed. i.e, strucnum\_in\_all.

strucnum\_in\_all = set.intersection(\*map(set, bridge\_arr\_dict.values()))

"""

strucnum\_in\_all = list(set.intersection(\*map(set, [b\_16, b\_17, b\_18, b\_19, b\_20, b\_21, b\_22, b\_23])))

# Sort the list so its contents will look more familiar to the user, i.e. be in numerical order.

strucnum\_in\_all = sorted(strucnum\_in\_all)

# Results in 9829 bridges starting with STRUCNUM = 01 0002 & ending with STRUCNUM = 58C0026.

"""

# End determine set of strucnum\_in\_all

# BEGIN Remove STRUCNUM not present in all dfs

for df\_name, df in filtered\_df\_nameToDF.items():

mask = df['STRUCNUM'].isin(strucnum\_in\_all)

df\_filtered = df[mask]

filtered\_df\_nameToDF[df\_name] = df\_filtered

# END Remove STRUCNUM not present in all dfs

"""

df2016CA = df2016CA[np.isin(df2016CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# No. of lines 48798

df2017CA = df2017CA[np.isin(df2017CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# No. of lines 49077

df2018CA = df2018CA[np.isin(df2018CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# 57830 lines long orig.

# No. of lines 49430

df2019CA = df2019CA[np.isin(df2019CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# 52036 lines long orig.

# No. of lines 49671

df2020CA = df2020CA[np.isin(df2020CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# 52619 lines long orig.

# No. of lines 50137

df2021CA = df2021CA[np.isin(df2021CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# 48562 lines long orig.

# No. of lines 50287

df2022CA = df2022CA[np.isin(df2022CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# 48562 lines long orig.

# No. of lines 51033

df2023CA = df2023CA[np.isin(df2023CA['STRUCNUM'].to\_numpy(), strucnum\_in\_all)]

# 48562 lines long orig.

# No. of lines 50929

"""

# Then to run a few checks, I make the STRUCNUM into sets for each newly modified dataframe strucnum\_2016\_mod, strucnum\_2017\_mod, etc.

# !!!

# Here

# !!!

# df\_names = ["df" + i.split('\_', 1)[0] for i in files]

strucnum\_mod = {'strucnum\_' + key[2:] + '\_mod': df['STRUCNUM'].unique() for key, df in filtered\_df\_nameToDF.items()}

"""

strucnum\_2016\_mod = df2016CA['STRUCNUM'].unique()

strucnum\_2017\_mod = df2017CA['STRUCNUM'].unique()

strucnum\_2018\_mod = df2018CA['STRUCNUM'].unique()

strucnum\_2019\_mod = df2019CA['STRUCNUM'].unique()

strucnum\_2020\_mod = df2020CA['STRUCNUM'].unique()

strucnum\_2021\_mod = df2021CA['STRUCNUM'].unique()

strucnum\_2022\_mod = df2022CA['STRUCNUM'].unique()

strucnum\_2023\_mod = df2023CA['STRUCNUM'].unique()

"""

# strucnum\_2017\_mod = strucnum\_2018\_mod = strucnum\_2019\_mod = strucnum\_2021\_mod = strucnum\_2020\_mod = strucnum\_2021\_mod = strucnum\_2022\_mod

from array import \*

# make lists of each set of strucnum to compare them.

strucnum\_mod = [ [value] for value in strucnum\_mod.values()]

"""

strucnum\_2016\_mod.tolist()

strucnum\_2017\_mod.tolist()

strucnum\_2018\_mod.tolist()

strucnum\_2019\_mod.tolist()

strucnum\_2020\_mod.tolist()

strucnum\_2021\_mod.tolist()

strucnum\_2022\_mod.tolist()

strucnum\_2023\_mod.tolist()

"""

# check that the STRUCNUM being pulled from each year are the same

# Is this the list of bridges common to all years that I would use to make the dfs that have the missing data I could replace? MVP II.

"""

if collections.Counter(strucnum\_2022\_mod) == collections.Counter(strucnum\_2021\_mod):

print ("The lists are identical")

else :

print ("The lists are not identical")

"""

#!!!

# 12-14-2023!!!

# BEGIN check content of the arrays of different years for bridges (strucnum) in each year procedure.

# Specifically, check the STRUCNUM in each array match each other - checking that the sets of bridges being separated out from the raw data for each year are the same.

def check\_array\_content(\*arrays):

if not arrays or len(arrays) < 2:

return True # If there are no or only one list, they are considered equal

for i in range(len(arrays)):

for j in range(i + 1, len(arrays)):

if not np.array\_equal(arrays[i], arrays[j]):

return False

return True

arrays\_to\_compare = [strucnum\_mod]

"""[strucnum\_2016\_mod, strucnum\_2017\_mod, strucnum\_2018\_mod, strucnum\_2019\_mod, strucnum\_2020\_mod, strucnum\_2021\_mod, strucnum\_2022\_mod, strucnum\_2023\_mod]"""

if check\_array\_content(\*arrays\_to\_compare):

print("All arrays are equal.")

else:

print("Arrays are not equal.")